

CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs

CP-Detector: ソフトウェア設定に関連する性能特性を用いた性能バグ検出

Haochen He
National University of Defense
Technology, China
hehaochen13@nudt.edu.cn

Erci Xu
National University of Defense
Technology, China
xuerci@nudt.edu.cn

Ji Wang
National University of Defense
Technology, China
wj@nudt.edu.cn

Zhouyang Jia*
National University of Defense
Technology, China
jiazhouyang@nudt.edu.cn

Tingting Yu
University of Kentucky
Lexington, KY, USA
tyu@cs.uky.edu

Xiangke Liao
National University of Defense
Technology, China
xkliao@nudt.edu.cn

Shanshan Li
National University of Defense
Technology, China
shanshanli@nudt.edu.cn

Yue Yu
National University of Defense
Technology, China
yuyue@nudt.edu.cn

1. 背景

- モダンなソフトウェアは、様々なシナリオに対応
多くの設定項目が存在
- ソフトウェアの性能に直接影響を与える設定項目も存在（リソースの割り当て変更など）
性能に関するバグの6割が、ソフトウェア内で設定に対応した適切な処理が
できていないことに起因[33]

以降このようなバグをConfiguration-handling Performance Bugs（CPBug）と呼ぶ

[33] Han, X., and Yu, T. An empirical study on performance bugs for highly configurable software systems. In International Symposium on Empirical Software Engineering and Measurement (ESEM) (2016).

1. 背景 | CPBugの例

sort_buffer_size (MySQLの設定項目) のCPBug

- ソートクエリ (ORDER BY) で利用するバッファサイズ
- バッファが大きいほど、より多くの結果をキャッシュ
ソートの性能が向上するはず…

→ **実際は**, ソートクエリ内にサブクエリが
含まれていると, 4.2倍**ソートの速度が低下**

- 原因

サブクエリでの結果のキャッシュが即解放される

- 修正方法

すぐに解放せずに以降のサブクエリで
バッファを再利用するようにする

MySQL Bug #21727

Reduced patch in sql/filesort.cc

// This code snippet is executed in each sub-
SELECT and the first 'if' statement only
get satisfied in the first sub-SELECT

```
1 - sort_keys = my_malloc  
    (sort_buffer_size, ...);  
2 + if(!table_sort.sort_keys)  
3 +   sort_keys = my_malloc  
    (sort_buffer_size, ...);  
4   ...
```

```
5 - x_free(sort_keys);
```

```
6 + if(!subselect)
```

```
7 +   x_free(sort_keys);
```

2. 関連研究

- CPBugに限らず，性能に関するバグは，通常のテスト方法ではfailしない
→ 検出が困難（処理は遅いがエラーは出ない）
- 性能バグの検出[28, 55, 67]
プロファイリングツールでボトルネックを検出
→ 検出されたボトルネックがバグであるかどうかは分からない[46]
- 非効率的なコードパターンを検出[46, 59, 61, 65, 66]
性能バグの検出は可能
→ 設定に関連している性能バグ（CPBug）かどうかは分からない

⇒ CPBugを検出することは困難

[28] Coppa, E., Demetrescu, C., and Finocchi, I. Input-sensitive profiling. In Conference on Programming Language Design and Implementation (PLDI) (2012).

[55] Shen, D., Luo, Q., Poshvanyk, D., and Grechanik, M. Automating performance bottleneck detection using search-based application profiling. In International Symposium on Software Testing & Analysis (ISSTA) (2015).

[67] Zaparanuks, D., and Hauswirth, M. Algorithmic profiling. In Conference on Programming Language Design and Implementation (PLDI) (2012).

[46] Nistor, A., Song, L., Marinov, D., and Lu, S. Toddler: Detecting performance problems via similar memory-access patterns. In International Conference on Software Engineering (ICSE) (2013).

[59] Song, L., and Lu, S. Performance diagnosis for inefficient loops. In International Conference on Software Engineering (ICSE) (2017).

[61] Su, P., Wen, S., Yang, H., Chabbi, M., and Liu, X. Redundant loads: A software inefficiency indicator. In International Conference on Software Engineering (ICSE) (2019).

[65] Yang, J., Subramaniam, P., Lu, S., Yan, C., and Cheung, A. How not to structure your database-backed web applications: a study of performance bugs in the wild. In International Conference on Software Engineering (ICSE) (2018).

[66] Yu, T., and Pradel, M. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In International Symposium on Software Testing & Analysis (ISSTA) (2016).

3. この研究の着眼点

- 設定に期待される性能（スループット，レイテンシ，実行時間など）に着目
- 先行研究では，ほとんどの性能バグが設定に関連している[36, 58]
- 一般的には，設定変更によってソフトウェアの性能が向上することを期待

⇒ 設定変更により期待される性能と実際の性能の不一致によって
CPBugを検出できる可能性

[36] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In Conference on Programming Language Design and Implementation (PLDI) (2012).
[58] Song, L., and Lu, S. Statistical debugging for real-world performance problems. In Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (2014).

4. 目的とアプローチ

目的

CPBugを検出するための自動テストフレームワークであるCP-Detectorを構築

アプローチ

1. テストオラクルを考えるために，CPBugを理解するための調査
2. ソフトウェアの各設定項目に対してテストケースを自動で作成
 - ・ 設定項目の性能特性の予測
 - ・ 設定に適用する値の戦略的なサンプリング

5. CPBugの調査 | 方法

調査項目

- ・ 設定変更によってCPBugを検出可能か？
- ・ どんなタイプの設定項目を，どのように変更させれば，CPBugが検出できるのか？
- ・ CPBugを引き起こす設定値にはどのような傾向があるか？
→ テストケースを効率よく生成するため

5. CPBugの調査 | 方法

調査対象ソフトウェア

- ・パフォーマンスが重要
- ・設定によって高度に動作をカスタマイズ可能
- ・現場で広く展開されているOSS

→ コミュニティによって注意深く保守され，CPBugの報告数が多いことが期待できる

- ・バグトラッカー，メーリングリスト，修正コミットから，CPBugを収集

1. “slow”, “long time”, “performance”などのキーワードで検索
2. 結果を目視しCPBugかどうかを判断

キーワード検索
のヒット数

Software	# KW†	# CPBugs	Software	# KW	# CPBugs
MySQL	398	35	Httpd	291	16
MariaDB	451	25	H2	17	1
MongoDB	425	26	Squid	4	1
PostgreSQL	14	5	Tomcat	25	3
RocksDB	16	3	GCC	618	39
Derby	18	3	Clang	192	16

計173個を収集

5. CPBugの調査 | 設定変更によって検出可能か？

収集したCPBugを目視

→ CPBugの大部分（87.6%）は、設定変更によって予期しない性能の変化があった

⇒ 設定変更により期待される性能と実際の性能の不一致によって、
CPBugの大部分を検出可能

5. CPBugの調査 | どんなタイプの設定項目を、どのように変更させれば、検出できるのか？

設定項目のタイプと変更の方向（設定値の増減やON/OFF）を目視で分類

設定項目のタイプ 設定値の変化の方向 期待される性能の変化 実際の性能の変化

PP-ID	Performance Properties (PP)				CPBugs	
	Configuration Option Type	Tuning Direction		Expected Performance Change	Actual Performance Change	# CPBugs (Pct.)
		Source Value	Target Value			
PP-1	Optimization on-off	OFF	ON	Rise (↑)	Drop (↓)	18 (12.0%)
PP-2	Non-functional tradeoff	Anti-performance [†]	Pro-performance [‡]	Rise (↑)	Drop (↓)	35 (23.3%)
PP-3	Non-functional tradeoff	Pro-performance	Anti-performance	Drop (↓)	More-than-expected drop (↓↓)	31 (20.7%)
PP-4	Resource allocation	Small	Large	Rise (↑)	Drop (↓)	24 (16.0%)
PP-5	Functionality on-off	ON	OFF	Rise (↑)	Drop (↓)	6 (4%)
PP-6	Functionality on-off	OFF	ON	Drop (↓)	More-than-expected drop (↓↓)	24 (16.0%)
PP-7	Non-influence option	Random	Random	Keep (→)	Drop (↓)	12 (8.0%)

以降，性能特性と呼ぶ

5. CPBugの調査 | CPBugを引き起こす設定値にはどのような傾向があるか？

収集したCPBugのうち，CPBugを再現できた27個の設定値を手動で網羅的に調査し，CPBugが発生する設定値を確認

グレー：バグが発生した数値の範囲

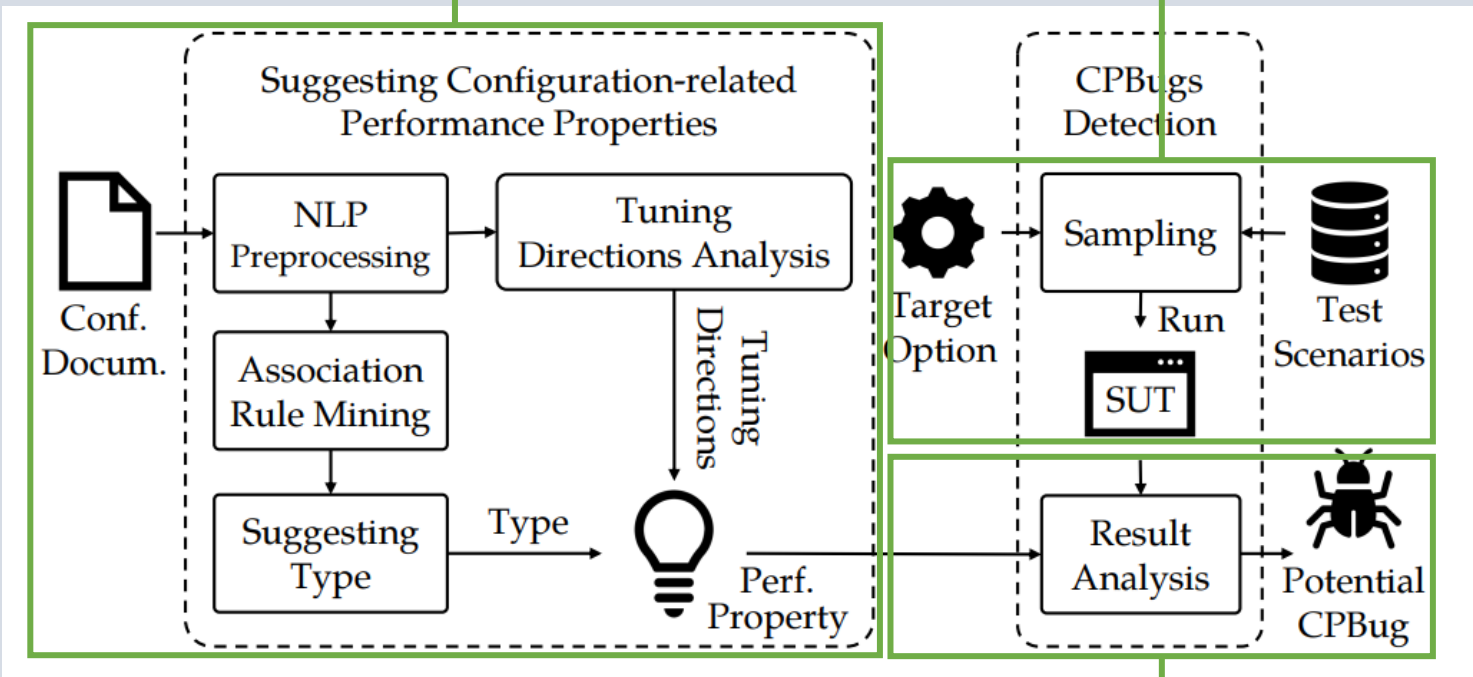
MySQL-21727	32K	48M	4G	MySQL-44723	8K	2G
MySQL-80784	5M	16G	2 ⁶⁴ -1	MySQL-78262	64K	64M
MySQL-62478	1M	16G	256G	MySQL-47529*	0	4M
MySQL-51325	5M	16G		MySQL-38551*	0	2M
Apache-54852	1	2	64	Apache-58037	0	1
Apache-48215	0	1	10 ³	Apache-50002	0	10 ³
MariaDB-13328	5M	16G		MariaDB-16283*	5M	128M
MariaDB-8696	64K	2M	64M	MariaDB-1212*	-1	1
MariaDB-12556	0	1	10 ³	MariaDB-145	0	3M
MariaDB-15016	1	64		MongoDB-17907*	256M	8G
MongoDB-20306	256M	12G	16G	MongoDB-24139*	256M	1G
RocksDB-122	256M	2G	16G	PostgreSQL-13750	1	2
PostgreSQL-15585	1	2	64	Squid-3189*	4K	6G

ほとんどの設定項目（26/27個）で，最大値 or 最小値の時にCPBugが発生
⇒ 設定項目の最大値 or 最小値をテストケースに使えば，効率よくCPBugを検出可能

6. CP-Detector

①各設定項目の性能特性を予測

②サンプリング戦略に基づいて
テストケースを生成，テスト実施



③性能特性を基に，期待した性能変化と実際の性能変化を比較
一致しなければ，設定項目と値をCPBugとして検出

6. CP-Detector | 設定項目の性能特性の予測

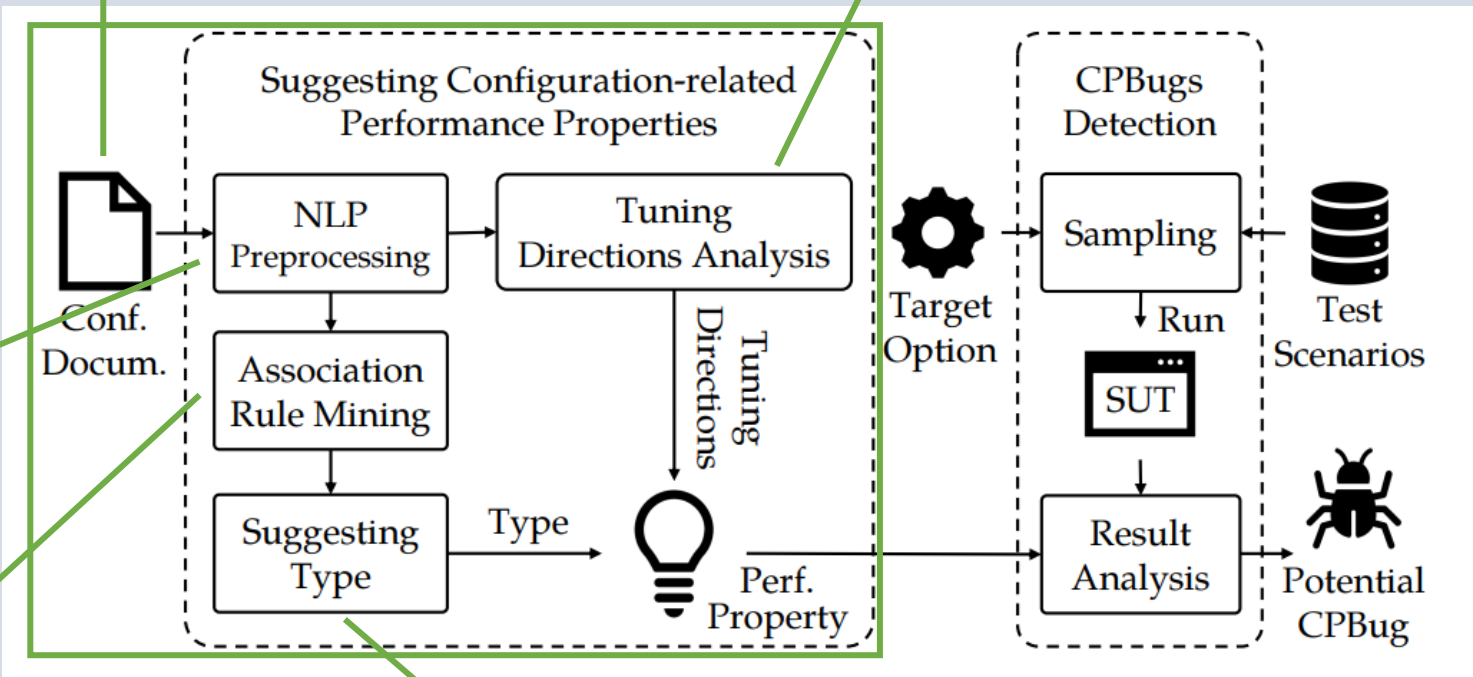
① 設定項目と紐づいた
ドキュメントを入力

⑤ CPBugを引き起こす設定値の変更方向も、
ドキュメントを基に予測

② 単語の正規化、
ドキュメントを
単語列として表現

③ 条件部：ドキュメントの単語列
結論部：ドキュメントが紐づいた
設定項目の性能特性
とし、ルール抽出

④ 予測時は、テストする設定項目の
ドキュメントを入力することで、
その設定項目の性能特性を予測



6. CP-Detector | 設定項目の性能特性の予測 | 評価 (RQ1)

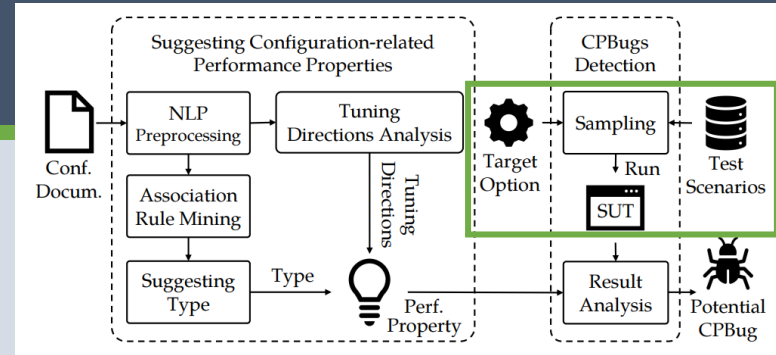
- 調査対象ソフトウェアから500個の設定項目をランダムサンプリング
- 設定項目を目視して性能特性を手動でラベル付けしたデータを，評価データとして利用

Type	# Option	Precision		Recall	
		CPD [†]	Base. [‡]	CPD	Base.
Resource	143	93.9%	69.4%	92.4%	93.2%
Tradeoff	84	70.7%	61.3%	66.9%	21.4%
Optimization	73	69.4%	27.3%	65.8%	18.8%
Functionality	100	82.2%	56.2%	55.6%	39.1%
Non-influence	100	90.1%	35.0%	67.1%	70.0%
Weighted Average		83.3%	52.4%	71.8%	54.8%

ベースラインモデルに比べて，
アソシエーションルールマイニングによる予測の方がPrecision， Recallともに高い

6. CP-Detector | テストケース生成

- 設定変更前後での性能を比較するため、設定変更前の設定値と後の設定値のペアを生成
 - 設定値がON/OFFの2値の場合、変更は2パターンのみ
 - 設定値が数値の場合、パターンは膨大
→ 戦略的にサンプリングする必要
- CPBugを調査結果：「設定項目の最大値 or 最小値を使えば大部分のCPBugを検出可能」
 - 変更前の設定値を最小値で固定，変更後の設定値を指数関数的に増加させる
例：<1, 2>, <1, 4>, <1, 8> ...
 - 変更後の設定値を最大値で固定，変更前の設定値を指数関数的に減少させる
例：<512, 1024>, <256, 1024>, <128, 1024> ...

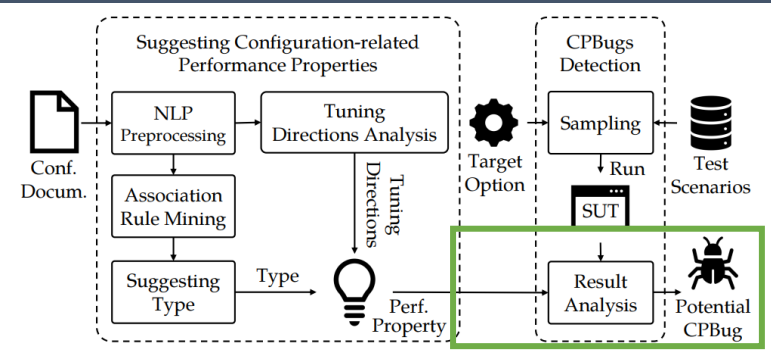


6. CP-Detector | CPBugの検出

- 性能特性に従って、テストケースによって設定値を変化させた際に、性能がどのように変化するかを確認
- 性能特性（リマインド）
例「最適化の設定をOFF → ONに変更したとき、性能が向上することを期待するが、実際には性能が低下する」

PP-ID	Performance Properties (PP)				CP Bugs	
	Configuration Option Type	Tuning Direction		Expected Performance Change	Actual Performance Change	# CP Bugs (Pct.)
		Source Value	Target Value			
PP-1	Optimization on-off	OFF	ON	Rise (↑)	Drop (↓)	18 (12.0%)
PP-2	Non-functional tradeoff	Anti-performance [†]	Pro-performance [‡]	Rise (↑)	Drop (↓)	35 (23.3%)
PP-3	Non-functional tradeoff	Pro-performance	Anti-performance	Drop (↓)	More-than-expected drop (↓↓)	31 (20.7%)
PP-4	Resource allocation	Small	Large	Rise (↑)	Drop (↓)	24 (16.0%)
PP-5	Functionality on-off	ON	OFF	Rise (↑)	Drop (↓)	6 (4%)
PP-6	Functionality on-off	OFF	ON	Drop (↓)	More-than-expected drop (↓↓)	24 (16.0%)
PP-7	Non-influence option	Random	Random	Keep (→)	Drop (↓)	12 (8.0%)

- 性能比：変更前の性能 / 変更後の性能 > Tr1 … (1)
 - 性能差：変更前の性能 – 変更後の性能 > Tr2 … (2)
- これら2つの式を満たせば、設定値を変化させた際に期待に反して性能が低下したことを意味する ⇒ CPBugとして検出



RQ2：既知，未知のCPBugをどの程度検出できるか？

RQ3：先行研究の性能バグ検出ツールと比較してどの程度の精度か？

RQ4：CP-DetectorのパラメータはCPBugの検出精度にどの程度影響するか？

6. CP-Detector | RQ2：既知，未知のCPBugをどの程度検出できるか？

既知のCPBug

PP Violated	# CPBugs	# Exposed	# FP
Optimization (PP-1)	7	5 [†] / 6 [‡]	2
Tradeoff-1 (PP-2)	12	10 / 12	0
Tradeoff-2 (PP-3)	9	7 / 8	0
Resource (PP-4)	12	11 / 12	1
Functionality-1 (PP-5)	2	1 / 2	0
Functionality-2 (PP-6)	9	5 / 8	2
Non-Influence (PP-7)	6	4 / 6	0
N/A	4	0 / 0	2
TOTAL	61	43 / 54	7

既知のCPBug数 検出できたCPBug数

未知のCPBug

17個の
未知のCPBugを検出
(内4つは誤検知)

Bug ID	Slowdown [‡]	Version(s)	Status
Clang #43576(1)	1.19× (E.T.)	v7 - latest	Confirmed
Clang #43576(2)	1.28× (E.T.)	v7 - latest	Confirmed
Clang #43084	2.9× (C.T.)	v3	Fixing
Clang #44359	1.2× (E.T.)	v7 - latest	Pending
Clang #44518	2.0× (C.T.)	v3 - latest	Fixing
GCC #91852	2.8× (C.T.)	v6 - latest	Pending
GCC #91895	4.4× (C.T.)	v4	Confirmed
GCC #91817	44× (C.T.)	v4	Confirmed
GCC #91875	1.85× (E.T.)	v7 - v8	Confirmed
GCC #93037	1.12× (E.T.)	v8 - latest	Pending
GCC #93521	2.52× (E.T.)	v8 - latest	Confirmed
GCC #93535	4.50× (E.T.)	v8 - latest	Confirmed
GCC #94957	hang (C.T.)	v7 - latest	Fixing

[‡] C.T.: Compiling Time, E.T.: Execution Time.

- コンパイル時間：2.8～44倍
 - バイナリの実行時間：1.19～1.85倍
- 何年も前のバージョンから、最新バージョンにかけて存在

- 先行研究の中で最も精度が高い性能バグ検出ツールの1つであるToddler[46]と比較
 - Toddler
非効率的なループによって引き起こされる性能バグを検出
 - Toddlerは、6 / 61個の既知のCPBugを検出
 - CP-Detectorは様々理由で引き起こされるCPBugを検出できるのに対し、Toddlerはループ内の非効率的な変数アクセスにのみ焦点を当てているため
- ⇒ CP-DetectorがToddlerよりも多くの種類の性能バグを検出可能

[46] Nistor, A., Song, L., Marinov, D., and Lu, S. Toddler: Detecting performance problems via similar memory-access patterns. In International Conference on Software Engineering (ICSE) (2013).

7. まとめ

背景

ソフトウェアの性能バグの多くは，ソフトウェアの設定項目に対して内部で適切な処理をしていないことで発生．性能バグは通常のテスト方法では検出できない．

着眼点，目的

設定変更前後で性能が期待通りに変化したかどうかに着目．

期待通りでないなら性能バグとして検出するテストフレームワークCP-Detectorを構築．

評価

- ・ 設定に関連する性能バグ（CPBug）に対して54 / 61個を検出可能
- ・ 17個の未知のCPBugを検出
- ・ 最新の性能バグ検出ツールと比較，CP-Detectorの方が多くの性能バグを検出可能

目視がいっぱい🙄

CPBug検出の先駆けとしての研究なので当然ではある

ただし、特にテストオラクルとなる性能特性の分類にあたって、

- ・元となるデータ数が173件と少ない
- ・手動で分類されていた

⇒ いくつかの部分を自動に置き換えるだけでも、
より広範なCPBugを検出できるのでは！